Mastering CSV in Ruby

Paweł Dąbrowski, Long Live Ruby

Table of contents

Introduction

CSV stands for Comma Separated Values, and it's a standard format for storing spreadsheet data. Modern internet systems often utilize this type of file to provide import and export functionality. Ruby makes it easy and enjoyable to parse those files in a modern, object-oriented way. This book is a perfect starting point for anyone who is not familiar with the CSV and those developers who work with CSV files daily but face some common problems and are looking for performance tips.

What you will learn by reading this book

After reading this book will know how to:

- parse CSV files from different sources to get the well-formatted data for your application
- create CSV files to create an importer functionality that will allow you to export the data from your application
- avoid performance issues when parsing large files
- deal with everyday problems like badly encoded or malformed files
- use Ruby gems that will speed up the development process and save you tons of time

This book covers all of the essential things about parsing CSV files with Ruby. It will be a from zero to hero journey.

Information about the Ruby version

I wrote the contents of the book based on the newest version of Ruby. I will constantly update it each time the latest version of Ruby will be available.

You don’t have to worry that some of the code snippets presented here will become outdated at some point in the future.The goal of this book is to always provide you the most relevant and valuable knowledge.

Latest date update: **28-04-2021**
Ruby version: **3.0.1**

Information about the author

My name is Paweł Dąbrowski, and I’m currently working as a CTO at iRonin, a software house with Ruby in the DNA. I’m also the founder of the Long Live Ruby blog. With over ten years of experience in Ruby and more than 13 years of experience in web development, I am passionate about learning and sharing the knowledge with other developers.

I work with CSV files daily when building complex data-processing applications that import and export tons of information. I believe that the knowledge about parsing CSV files is valuable for every developer working with Ruby, and that kind of book will be your best friend.

The anatomy of CSV format

Before we get our hands dirty by writing Ruby code, it’s good to learn something more about the CSV format itself. Such an introduction always gives us a better understanding of the topic and makes us smarter even when programming with other languages.

The structure of the CSV file

Imagine a simple table with information about the users:

First name	Age	Location
Tim	33	Berlin
Kate	29	New York

If we would like to export the table to the CSV format, it could look the following:

first_name,age,location
Tim,33,Berlin
Kate,29,New York

By looking at the above example, we can quickly write down the critical elements of the CSV format:

- The file starts with the line where headers are listed
- Each file entry is separated with a new line
- Columns are usually separated by the , or ; character

When there is a possibility that the columns' values will contain , or ; characters, then values are usually wrapped in double quotes to avoid parsing problems.

Where CSV files are used

CSV is the default format for spreadsheets. A spreadsheet is a file with some structured data where all records share the same headers, meaning that they belong to the same group. A good example is the list of the employees for a given company. Probably each record would contain the name and e-mail address of the employee.

CSV format is used for files everywhere where there is a need to import some data into the system or export the data from the system. Such a file is also human-readable so that we can create it by hand. An excellent example of the system that exports data in the CSV format is the online banking system that allows you to export your transactions. You can then import such a file to software that helps you keep tracking your expenses or grow your savings.

Other places where you can meet files in the CSV format:

- Trading platforms that allow you to export the list of transactions
- Mail clients that allow you to export the contact or the list of contacts (or import as well)
- Enterprise systems that handle the employee data for the company

Common problems with parsing CSV files

Even though the CSV format is quite simple, you can run into some problems when parsing files in that format:

- **Parsing large files** - usually, CSV files are pretty small, but sometimes a single file can weigh even a few gigabytes. Opening and parsing such files can be a tricky thing. I will cover that topic later and give you some tips and good practices that you can use later to make it less painful and more performant.
- **Badly formatted files** - most of the time, you will have to work with files generated automatically. Still, even those can contain some poorly formatted values that can make the parsing harder.
- **Different encoding** - sometimes it happens that you receive values with a different encoding than you expected. Things might get a little bit difficult if you have to transform the characters into a readable format or remove the invalid part without touching the valid one.

The problems I mentioned above are pretty common and are reported many times around the internet. The solution for them is not that hard to implement, but it's sometimes time-consuming to find the exact fix, so I will guide you to fix those problems as well.

How Ruby handles files in CSV format

The standard Ruby library provides a robust and straightforward interface for parsing and creating files in CSV format. Most of the time, you won't need any external libraries.

You might look for some more advanced solutions if you would like to have more flexibility in creating the CSV files, checking the differences between two CSV files, or accessing a file's contents in a non-standard way. Thankfully, the Ruby community already provided a bunch of Ruby gems to achieve all of this.

Parsing CSV file

Before we start writing some code, let's prepare a test CSV file that we will be working on. Visit https://longliveruby.com/books/mastering-csv-files/file.csv and download a sample CSV file. I will use its contents to show you how you can quickly parse the data.

Parsing files from different sources

In most cases, there are three types of ways that you can get the CSV content into your code. I will quickly go through them to show you how to pull the data quickly.

Parsing CSV from a variable

```
require 'csv'

# Parse from variable
csv_string = "header1,header2\nvalue1,value2"
CSV.parse(csv_string)
# => [["header1", "header2"], ["value1", "value2"]]
```

As you can see, you can simply pass the variable that contains the data in CSV format and use `CSV.parse` to get the formatted output.

Parsing CSV from a file

```
require 'csv'

# Read directly from file
file_path = './file.csv'
CSV.read(file_path)
# => [["header1", "header2"], ["value1", "value2"]]
```

When using `CSV.read` way, you can pass the file path directly. The explicit way of reading the file is to get the contents of a file first and then pass it to the CSV:

```
# Read explicitly from a file
file_path = './file.csv'
csv_string = File.read(file_path)
CSV.parse(csv_string)
# => [["header1", "header2"], ["value1", "value2"]]
```

However, the first way of reading the file seems handier.

Parsing CSV from a URL

```
require 'csv'
require 'open-uri'

# Read from URL
url =
'https://longliveruby.com/books/mastering-csv-files/file.cs
v'
CSV.parse(URI.parse(url).open)
# => [["header1", "header2", "header3"], ["value1a",
"value1b", "value1c"], ["value2a", "value2b", "value2c"]]
```

The only difference between this example and those two first is that we open the URL to get the web page contents and then pass it directly to the CSV module's parse method. Simple as that.

Parsing large files

The file we tested in the above examples is tiny. We need something bigger:

```
# Create a big file
headers = ['id', 'first_name', 'last_name', 'location',
'age']

CSV.open('bigfile.csv', 'w', write_headers: true, headers:
headers) do |csv|
 10_000_000.times do |i|
   csv << [i, "Joh#{i}", "Doe#{i}", "New York", "35"]
 end
end
```

The `bigfile.csv` weighs around 400MB; it took the program around 30 seconds to create it and fill it with the data. Now, we have something with which we can begin making accurate tests.

Using `CSV.read` or `CSV.parse` is a bad idea as it will blow up your program's memory. The most optimal solution for large files is to use `CSV.foreach`:

```
CSV.foreach('./bigfile.csv', headers: true) do |row|
 # do something with row['first_name'] or row['age']
end
```

The foreach method, instead of loading the whole file into memory, iterates over the file row by row. Always use it when parsing large files as you get access to the rows immediately and do not load everything into memory.

Summary about CSV and parsing modes

Before we jump into the next chapter, let's quickly summarize the ways we can use the CSV module to load the data:

- **Iterating over every row of the file** - we can achieve that by using `CSV.foreach`. It's a perfect solution for parsing large files as we immediately access the data and do not load everything into memory.
- **Reading and parsing the file** - we can achieve that by using `CSV.read`. This way is a good solution for smaller files that we don't want to open explicitly.
- **Parsing the data** - if the CSV data is located in the variable, we can parse it using the `CSV.parse` method.

Parsing the table and rows separately

In many cases, when parsing CSV data, we don't want to just receive a multi-dimensional array due to parsing. Thankfully we are not limited to accessing the rows by using only indexes.

Accessing CSV rows as a hash

Instead of an array of values, we can access the given row as a hash where the header names as keys and values as… values. We just have to pass the `headers: true` option:

```
csv_string = "header1,header2\nvalue1,value2"
CSV.parse(csv_string, headers: true).map do |row|
 row['header1']
end
# => ['value1']

file_path = './file.csv'
CSV.read(file_path, headers: true).map do |row|
 row['header1']
end
# => ['value1']
```

When you are using CSV with the headers option, you receive `CSV::Table` instance, which contains multiple `CSV::Row` instances for each row instead of an array.

Manipulating the single row

When we are receiving each row as a `CSV::Row` instance we get a lot of flexibility. Let's start with creating the instance that we can play with:

```
row = CSV::Row.new(["first_name", "last_name", "age"],
["John", "Doe", "35"])
```

As I mentioned before, you can access a row as a hash but when you need a hash representation then you can simply call `to_h` on a row:

```
row.to_h
=> {"first_name"=>"John", "last_name"=>"Doe", "age"=>"35"}
```

You can also iterate of each pair:

```
row.each_pair.map do |header, value|
 [header, value]
end
# => [["first_name", "John"], ["last_name", "Doe"], ["age",
"35"]]
```

Manipulating the whole table

I want to show you one more cool thing that you can do with a complete CSV table. If you want to quickly get values only for given columns you can use the `values_at` method:

```
csv =
"first_name,last_name,age\nJohn,Doe,35\nTim,Doe,20\nHelen,D
oe,30"
CSV.parse(csv, headers: true).values_at('first_name',
'last_name')
# => => [["John", "Doe"], ["Tim", "Doe"], ["Helen", "Doe"]]
```

If you would like to get only values for one column, first_name, for example, you can do without any effort:

```
CSV.parse(csv, headers: true)['first_name']
=> ["John", "Tim", "Helen"]
```

Creating CSV files

The simplest representation of CSV in code is a multi-dimensional array where the first element (first array) is the list of headers. Having that in mind, we can easily create the CSV file from any data.

The main rule is to push array headers first and then, for every record we want to include in CSV, push arrays with values:

```
headers = ['first_name', 'last_name', 'age']
User = Struct.new(:first_name, :last_name, :age)

users = [
 User.new('John', 'Doe', '33'),
 User.new('Tim', 'Doe', '25'),
 User.new('Tina', 'Doe', '30')
]

CSV.open('./users.csv', 'w') do |csv|
 csv << headers
 users.each do |user|
   csv << [user.first_name, user.last_name, user.age]
 end
end
```

I used the `CSV.open` in the write-only mode, so the error is not raised if the file is not existing yet. Also, when the file with the given name exists, the code overwrites it. You can check now the users.csv file, and you will see that we create a standard file in CSV format with headers and three rows.

If you would like to make your code more readable, we can use `add_row` method which is an alias for `<<` but it looks more readable:

```
CSV.open('./users.csv', 'w') do |csv|
 csv.add_row(headers)
 users.each do |user|
   csv.add_row([user.first_name, user.last_name, user.age])
 end
end
```

Editing existing files

Similar to the creation process, we can make usage of the `CSV.open` method. The only difference is the mode flag. This time instead of the write-only mode, I will use read and write mode - a+. With such mode, the existing file is opened, and the pointer is set to the last row so we can add more rows without losing the original content:

```
require 'csv'

headers = ['first_name', 'last_name', 'age']
User = Struct.new(:first_name, :last_name, :age)

users = [
 User.new('John', 'Doe', '33'),
 User.new('Tim', 'Doe', '25'),
 User.new('Tina', 'Doe', '30')
]

CSV.open('./users.csv', 'w') do |csv|
 csv.add_row(headers)
 users.each do |user|
   csv.add_row([user.first_name, user.last_name, user.age])
 end
end

new_user = User.new('Jenny', 'Doe', '23')
CSV.open('./users.csv', 'a+') do |csv|
 csv.add_row([new_user.first_name, new_user.last_name,
new_user.age])
end
```

Modifying existing rows

If you would like to edit an existing row in a CSV file, you have to collect all rows from the file, find the row you want to update, update it, and then rewrite the whole file.

If we would like to change the age of Jenny Doe, from 23 to 25, we can do this with the following code:

```
rows = CSV.open('./users.csv', headers: true).map(&:to_h)
row = rows.find { |r| r['first_name'] == 'Jenny' }
row_index = rows.index(row)
row['age'] = '25'
rows[row_index] = row

CSV.open('./users.csv', 'w') do |csv|
 csv.add_row(rows.first.keys)
 rows.map { |row| csv.add_row(row.values) }
end
```

More advanced CSV manipulation

The usage examples I presented in the last chapters were relatively simple, so it's time to learn something more advanced and less commonly presented.

Preprocessing the data

Let's assume that we have the following file named users.csv with the following contents:

first_name,last_name,age
John,Doe,19
Tim,Doe,25

Now, let's load the contents into our code:

```
csv = CSV.parse(File.read('./users.csv'), headers:
:first_row, return_headers: false)
csv.first['age']
# => "19"
```

We get the age value as a string instead of the integer. To get the integer we have to modify our code and use converter:

```
csv = CSV.parse(File.read('./users.csv'), headers:
:first_row, return_headers: false, converters:
[CSV::Converters[:integer]])
csv.first['age']
# => 19
```

Everything works as expected. By default, we have the following converters available:

- :integer
- :float
- :numeric
- :date
- :date_time
- :all

You can always check the complete list by calling `CSV::Converters.keys`

Creating own preprocessor (converter)

Writing a new preprocessor is easy as the converter is just a lambda expression. To confirm this, you can select one of the converters and invoke the call method with an argument on it:

```
CSV::Converters[:float].call("13.1")
# => 13.1
```

Let's play with the code a little bit and create an Email value object which parses the email addresses and provides two methods: domain and username:

```
class Email
 def initialize(value)
   @value = value
 end

def to_s
  @value
end

 def domain
   @value.split('@').last
 end

 def username
   @value.split('@').first
 end
end
```

We can now make an email converter that would replace any email address with the `Email` object:

```
csv =
"first_name,email\nJohn,john@doe.com\nTim,tim@doe.com"
CSV::Converters[:email] = ->(value) { value.include?('@') ?
Email.new(value) : value }
parsed_csv = CSV.parse(csv, headers: :first_row,
converters: [:email])
parsed_csv.first['email'].domain
# => doe.com
```

There are two essential rules when it comes to creating converters:

- ensure that you will always return a value from the converter
- declare one argument if you want to parse the only value, declare two arguments if you would like to parse additional information about the given value

I mentioned the second argument - additional information about the given value. Let's take a closer look at it:

```
csv =
"first_name,email\nJohn,john@doe.com\nTim,tim@doe.com"
conv = ->(arg1, arg2) { [arg1, arg2] }
parsed_csv = CSV.parse(csv, headers: :first_row,
converters: [conv])
parsed_csv.first['email']
# => ["john@doe.com", #<struct CSV::FieldInfo index=1,
line=2, header="email">]
```

Now we have access to the struct that contains the index, line, and header of the given field. It provides us with a lot of flexibility.

Ruby on Rails and CSV format

Before we generate some CSV files from the Rails application, we have to create it first and some of the test data. Let's do it quickly now:

```
rvm use ruby-3.0.1
nvm use 12.13.0
gem install rails
rails new csvapp -d=mysql
```

In the above snippet, I used RVM to use the newest Ruby version, NVM tool to select the Node.js version, and then I installed Rails gem and generated a new project with the support for the MySQL database. After a few seconds, the project is ready, and we can develop the first model.

```
cd csvapp/
rake db:create
bin/rails generate scaffold User first_name:string
last_name:string email:string age:integer
rake db:migrate
rails s
```

Our app is ready. I generated a simple scaffold for the User model so you can now access localhost:3000/users/new address and add some new users; we will need them later to manipulate their data.

I added the following users:

- John Doe, john@doe.com, age 25
- Tim Doe, tim@doe.com, age 30

- Tina Doe, tina@doe.com, age 29

Exporting the data in CSV format from the controller level

The goal now is to be able to download the CSV list of users when accessing localhost:3000/users.csv. Create the proper view first:

```
touch app/views/users/index.csv.erb
```

now we have to update the `UsersController` controller and the `index` action to respond to the CSV format properly:

```
def index
 @users = User.all

 respond_to do |format|
   format.html
   format.json
   format.csv do
     headers['Content-Disposition'] = "attachment;
filename=\"users.csv\""
     headers['Content-Type'] ||= 'text/csv'
   end
 end
end
```

After visiting the localhost:3000/users.csv address, the browser sends a blank users.csv file. The last step is to update the view. To make the logic more separated, I will implement the `to_csv` method for the User model:

```
require 'csv'

class User < ApplicationRecord
 def self.csv_headers
   %w(first_name last_name age)
 end

 def to_csv

::CSV.generate_line(attributes.slice(*self.class.csv_header
s).values)
 end
end
```

The `to_csv` method utilizes `CSV.generate_line`, which simply accepts an array of values and returns the CSV row, a string where the values are separated by a comma, and there is a new line character at the end of the string.

In the view the first step is to render headers and iterate over the `@users` array and call `to_csv` on each record:

```
<%= CSV.generate_line(User.csv_headers) -%>
<% @users.each do |user| %>
 <%= user.to_csv -%>
<% end %>
```

I used `<%= … -%>` to avoid a blank line between each record as it would break our CSV with empty rows. Visit localhost:3000/users.csv again, and now the CSV with data will be downloaded.

Exporting records in the CSV format

You might need to be able to export the records from a given model to the CSV format. It's good to keep the logic somewhere and not repeat it for every model in our codebase. To stay DRY will create a model concern:

```
touch app/models/concerns/csv_exportable.rb
```

I will ignore `created_at` and `updated_at` columns as in most cases; we wouldn't want to export them:

```
module CsvExportable
 extend ActiveSupport::Concern

 class_methods do
   def export_to_csv(file_path)
     ignore_columns = %w(created_at updated_at)
     headers = self.column_names - ignore_columns

     CSV.open(file_path, 'w') do |csv|
       csv.add_row(headers)
       self.find_each do |record|
         csv.add_row(record.attributes.values_at(*headers))
       end
     end

     true
   end
 end
end
```

You can now include the module in our `User` model:

```
class User < ApplicationRecord
 include CsvExportable
end
```

Now, you can simply call `User.export_to_csv('./file.csv')` to export all records from the database. Because I used the `find_each` method, you don't have to worry about the big size of your table as the records will be fetched in batches.

Loading records from the CSV format

Similar to the previous step, where we were exporting records from the database, I will also create a model concern to make the logic more reusable across other models. Let's start with creating the file:

```
touch app/models/concerns/csv_importable.rb
```

We also need some file with the data to ensure that our code will work well with the CSV files. You can save the following rows in the file named users.csv:

first_name,last_name,email,age,location
John,Doe,john@doe.com,25,New York
Tim,Doe,tim@doe.com,30,Berlin
Tina,Doe,tina@doe.com,29,Barcelona

I intentionally added the location column though we don't have it in our `User` model. This column will help us test if we don't try to pass an attribute that is not present in our model.

We can now open the `app/models/concerns/csv_importable.rb` file and create the class method that can be easily included in any model to extend the class by the functionality of importing records from the CSV file.

The method will be pretty simple but consist of the following three steps:

- Reading the rows from the `file_path` passed as the first argument
- Filtering each row values to select only those that exist as the attributes in the given model
- Creating new record using the filtered attributes

Let's make it happen:

```
module CsvImportable
 extend ActiveSupport::Concern

 class_methods do
   def import_from_csv(file_path:, columns:)
     records = []

     CSV.foreach(file_path, headers: true).each do |row|
       attributes = row.to_h.slice(*columns)
       records << create!(attributes)
     end

     records
   end
 end
end
```

Because I used the `create!` method, when one of the records is not saved, an error will be raised, and the process will be stopped. You can even wrap the whole loop in the transaction to ensure that we will create all records or none if they are invalid.

Since I created the above code only for demonstration purposes, I won't spend more time refactoring it as it does what it should quite well. We can now include the concern into our User model:

```
require 'csv'

class User < ApplicationRecord
 include CsvImportable
end
```

and test it in the console:

```
User.import_from_csv(file_path: './users.csv', columns:
%w[first_name last_name email age])
# => [#<User:0x00007fc911c76fa0...]
```

Useful Ruby gems

Dealing with CSV with Ruby is relatively straightforward, but some valuable ruby gems were created. In this section, I will mention two of them I consider the most useful.

Using external libraries for CSV can speed up the development. Still, you can also successfully write your code for most cases of processing CSV with Ruby using the knowledge from this book.

Checking the changes between two CSV files

I work a lot with large CSV files that contain employees data. Companies often update the data in the system by importing such files frequently.

Doing the full update each time the file is sent to the system might be a time-consuming process depending on the file size.
The idea to improve the process is to spot the changes, deletions, and additions between the current file and previous file, so there is no need to touch the records that didn't change at all (in my case, most of the records are the same).

To achieve that, you can use the csv-diff gem. Let me show you how it works in practice. The first step is to install it:

gem install csv-diff

Because we will check the differences between two files, let's create some test data that we can use. Create the users.csv file with the following rows:

uid,first_name,last_name,age
1,John,Doe,19
2,Tim,Doe,25

and the file named updated_users.csv that represents the file with the updated contents:

uid,first_name,last_name,age
1,John,Doe,20
3,Rick,Doe,30

We can now open the ruby console, load the gem and check the differences between those files:

```
require 'csv-diff'

diff = CSVDiff.new('./users.csv', './updated_users.csv')
diff.summary
# => {"Delete"=>1, "Update"=>1, "Add"=>1}
```

The library correctly detected that we deleted the row with Tim Doe, added a row for Rick Doe, and updated the age of Time Doe. The general information is helpful, but often we need a deeper insight into the changes that were made by the system.

Checking deletions

```
diff.deletes
# => {"2"=>#<CSVDiff::Algorithm::Diff:0x00007fb5f28539f0
@diff_type=:delete, @fields={"uid"=>"2",
"first_name"=>"Tim", "last_name"=>"Doe", "age"=>"25"},
@row=2, @sibling_position=2>}
```

In our case, the uid column is the unique column, but you can configure it how you want. When the column is not explicitly specified, the library takes the first column, which is the id in most cases.

For the deleted rows, we can check the contents and the position of the row as well.

Checking additions

```
diff.adds
# => {"3"=>#<CSVDiff::Algorithm::Diff:0x00007fb5f2852848
@diff_type=:add, @fields={"uid"=>"3", "first_name"=>"Rick",
"last_name"=>"Doe", "age"=>"30"}, @row=2,
@sibling_position=2>}
```

Like in the case of deletions, we can also check the attributes and the position of the row for additions.

Checking changes

```
diff.updates
# => {"1"=>#<CSVDiff::Algorithm::Diff:0x00007fb5f2852dc0
@diff_type=:update, @fields={"uid"=>"1", "age"=>["19",
"20"]}, @row=1, @sibling_position=1>}
```

When it comes to updates, the library provides information about updated rows. It's super useful for the system where you have to prepare a detailed notification of the changes or do some actions only when the given field is changed.

The next steps

The gem also informs the developer when the unique keys are duplicated, or one of the files does not contain the same type of columns. To check the warning feature and some additional configuration possibilities, please check the official library repository: https://github.com/agardiner/csv-diff

Processing CSV files from Smarter CSV library

If you don't want to write your code for reading and processing CSV files, there is a handy and robust library out there. It's called Smarter CSV and provides a set of methods that will make your life easier.

Start with installing it in your system:

```
gem install smarter_csv
```

In the previous example, we created the users.csv file with the following contents:

```
uid,first_name,last_name,age
1,John,Doe,19
2,Tim,Doe,25
```

Let's use it to see how the Smarter CSV library processes it:

```
require 'smarter_csv'

SmarterCSV.process("./users.csv")
# => [{:uid=>1, :first_name=>"John", :last_name=>"Doe",
:age=>19}, {:uid=>2, :first_name=>"Tim", :last_name=>"Doe",
:age=>25}]
```

The library automatically parses the integer values and returns an array of hashes where the column names are symbols. This is a beneficial behavior expected by most developers when parsing the CSV file.

It's also possible to validate the existence of the header. If the header is not available in the passed file, the library will throw an error:

```
SmarterCSV.process("./users.csv", required_headers:
%i[first_name email])
# => SmarterCSV::MissingHeaders (ERROR: missing headers:
email)
```

The library's documentation is well-written to quickly learn about most of the features that the gem providers. If you liked the snippets I provided above, here is the documentation where you can learn about other configuration options: https://github.com/tilo/smarter_csv/wiki

Fixing common problems

I have been working with CSV files for the last few years, and I typically was dealing with some common errors that are well-known in the Ruby community. I decided to create this section to save you some time searching through StackOverflow or Google and answer the problematic situation that you can run into when parsing CSV files with Ruby.

If I missed any case you recently dealt with, please contact me, and I'm happy to update this section as soon as possible.

High memory usage

Typically, the memory usage level is not a problem when parsing CSV files, but it can become more problematic as you parse large files. You should avoid loading all the records into memory at once. I always suggest using the `CSV.foreach` method, which iterates over every row instead of loading the whole file into the memory and then processing.

Imagine how this approach can speed up a process where you have to find only one record among millions of other records or insert those records in the database.

Parsing file with the duplicated headers

Sometimes, the CSV file contains duplicated headers, and a row can have different values for each occurrence. It is good to know what to do in such a case.

Standard library

When you are using the standard Ruby library and methods like `CSV.parse`, `CSV.read` or `CSV.foreach`, the file will be parsed without the errors but only the first occurrence of the given header will be taken into the account:

```
csv = "first_name,email,first_name\nTom,tom@gmail.com,Jim"
CSV.parse(csv, headers: true).first['first_name']
# => Tom
```

Ruby sees the duplication but ignores it when generating output

Smarter CSV library

In case of the Smarter CSV library an error would be raised if the duplicated headers will be present in the file:

```
require 'smarter_csv'

SmarterCSV.process("./users.csv")
# => SmarterCSV::DuplicateHeaders (ERROR: duplicate
headers: first_name,first_name)
```

Collecting duplicate headers

If you don't want to ignore the duplications, the standard CSV library allows you to take a row and use `each_pair` method on it. With that method you can collect the duplications and process it further if needed:

```
rows = []
csv = "first_name,email,first_name\nTom,tom@gmail.com,Jim"
CSV.parse(csv, headers: true).each do |row|
  attributes = {}
  row.each_pair do |column, value|
    attributes[column] ||= []
    attributes[column] << value
  end

  rows << attributes.transform_values { |v| v.size == 1 ?
v.first : v }
end

rows
# => [{"first_name"=>["Tom", "Jim"],
"email"=>"tom@gmail.com"}]
```

Dealing with the encoding issues

Let's consider the following case: we are processing the CSV file with the custom encoding where one of the values contains some special characters:

```
uid,first_name,last_name,age
1,John,Non spécifié,19
2,Tim,Doe,20
3,Marcel,Doe,30
```

John's last name is not specified, and the value is in french. The file encoding is ISO 8859-1. Opening it the standard way will throw an error:

```
CSV.read("./users.csv")
# => Invalid byte sequence in UTF-8 in line 2.
(CSV::MalformedCSVError)
```

To fix the code, we have to define the encoding explicitly:

```
CSV.read("./users.csv", encoding: "ISO-8859-1", headers:
true).each do |row|
  puts row['last_name']
end

# => Non spécifié
# => Doe
# => Doe
```

Parsing file with a non-standard separator

Usually, values in the CSV file are separated by the comma or semicolon. By the default, the standard Ruby library expects comma but you can overwrite this configuration and use any separator you want:

```
csv = "first_name|last_name\nJohn|Doe"
CSV.parse(csv, col_sep: "|", headers: true).first.to_h
# => {"first_name"=>"John", "last_name"=>"Doe"}
```

The key is to use the col_sep setting.

Parsing file with duplicated rows

Let's assume that we are parsing the following file named users.csv:

```
uid,first_name,last_name,age
1,John,Doe,19
2,Tim,Doe,20
1,John,Doe,19
3,Marcel,Doe,30
```

The row for John Doe is presented twice in the processed file. You can distinct rows as an array by using the `uniq` method:

```
CSV.read("./users.csv", headers: true).to_a.uniq
```

With the SmarterCSV gem you have to do the same:

```
SmarterCSV.process("./users.csv").uniq
```

In both cases you receive a simple structure so to create the CSV objects again you have to do the parsing again:

```
records = CSV.read("./users.csv", headers: true).to_a.uniq
csv = records.map { |r| r.join(",") }.join("\n")
CSV.parse(csv, headers: true)
```

Parsing files with empty rows

In the standard library, you can automatically remove empty rows by passing the `skip_blanks: true` option:

```
CSV.read("./users.csv", headers: true, skip_blanks: true)
```

The Smarter CSV gem will remove the blank lines automatically.

Standard CSV library cheat sheet

Parsing modes

- `CSV.parse` - the method accepts a string which is data in CSV format

- `CSV.read` - the method accepts a path to CSV file, loads data into memory, and parses it
- `CSV.foreach` - the method accepts a path to a CSV file and returns an enumerator. It does not load the whole into the memory but allows you to iterate over each row
- `CSV.parse_line` - the method accepts a string which is a single line from CSV data

Available options

Information is copied from the official documentation.

- `row_sep` - Specifies the row separator; used to delimit rows.
- `col_sep` - Specifies the column separator; used to delimit fields.
- `quote_char` - Specifies the quote character; used to quote fields.
- `field_size_limit` - Specifies the maximum field size allowed.
- `converters` - Specifies the field converters to be used.
- `unconverted_fields` - Specifies whether unconverted fields are to be available.
- `headers` - Specifies whether data contains headers, or specifies the headers themselves.
- `return_headers` - Specifies whether headers are to be returned.
- `header_converters` - Specifies the header converters to be used.
- `skip_blanks` - Specifies whether blanks lines are to be ignored.
- `skip_lines` - Specifies how comments lines are to be recognized.
- `strip` - Specifies whether leading and trailing whitespace is to be stripped from fields.

- `liberal_parsing` - Specifies whether CSV should attempt to parse non-compliant data.
- `nil_value` - Specifies the object that is to be substituted for each null (no-text) field.
- `empty_value` - Specifies the object that is to be substituted for each empty field.